Enriching software architecture documentation

Jansen, Anton; Avgeriou, Paris; Ven, Jan Salvador van der

Published in:
Journal of Systems and Software

DOI:
10.1016/j.jss.2009.04.052

Publication date:
2009

The Journal of Systems and Software 82 (2009) 1232–1248

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Enriching software architecture documentation

Anton Jansen *, Paris Avgeriou, Jan Salvador van der Ven [1]

Department of Mathematics and Computing Science, University of Groningen, P.O. Box 800, 9700AV Groningen, The Netherlands

ARTICLE INFO

Article history:
Received 1 August 2008
Received in revised form 10 April 2009
Accepted 19 April 2009
Available online 8 May 2009

Keywords:
Architectural knowledge
Software architecture
Design decisions
Documentation
Design rationale

ABSTRACT

The effective documentation of Architectural Knowledge (AK) is one of the key factors in leveraging the paradigm shift toward sharing and reusing AK. However, current documentation approaches have severe shortcomings in capturing the knowledge of large and complex systems and subsequently facilitating its usage. In this paper, we propose to tackle this problem through the enrichment of traditional architectural documentation with formal AK. We have developed an approach consisting of a method and an accompanying tool suite to support this enrichment. We evaluate our approach through a quasi-controlled experiment with the architecture of a real, large, and complex system. We provide empirical evidence that our approach helps to partially solve the problem and indicate further directions in managing documented AK.

1. Introduction

The knowledge about a software architecture and its environment is called Architectural Knowledge (AK) (Kruchten et al., 2006) and has resulted in a paradigm shift in the software architecture community (Lago and Avgeriou, 2006; Avgeriou et al., 2007, 2008). The most important type of AK are architectural (design) decisions, which shape a software architecture (Jansen and Bosch, 2005). Other types of AK include concepts from architectural design (e.g. components, connectors) (Tang et al., 2007), requirements engineering (e.g. risks, requirements), people (e.g. stakeholders and roles), and the development process (e.g. activities) (de Boer et al., 2007).

There is a growing awareness both in industry and academia that effectively sharing AK, both inside the developing organization and with external actors, is one of the key factors for project success (Lago and Avgeriou, 2006; Avgeriou et al., 2007, 2008). Organizations are already exploring this new paradigm by conducting research on the benefits of knowledge-based architecting (Lago et al., 2008). The aim of this research is to bring enough evidence to convince the relevant stakeholders to embrace this new way of working by producing and consuming documented AK. In specific, stakeholders need to spend significant effort in documenting the AK, and therefore must be convinced that they will get a good return on their investment. On the other hand, when consuming AK, stakeholders need to trust the credibility of the documented knowledge (e.g. maintainers should have confidence in how up-to-date the AK is).

Documenting AK is not new, but has been common practice in the software architecture community over the last years (Clements et al., 2002). In both heavyweight processes (e.g. the Rational Unified Process (Kruchten, 2000)) and agile processes (e.g. XP, SCRUM (Beck and Fowler, 2000; Schwaber and Beedle, 2001)), knowledge is documented to facilitate communication between stakeholders. The essential difference between the former and the latter is that heavyweight processes determine large documents up front, while agile processes produce less documentation, strictly when needed. In essence, the knowledge in both cases is transformed from implicit or tacit knowledge (Nonaka and Takeuchi, 1995) into explicit knowledge (Hansen et al., 1999). Two types of explicit knowledge can be discerned: documented and formal knowledge. Documented knowledge is expressed in natural language and/or images, while formal knowledge is expressed in formal languages or models with clearly specified semantics (e.g. ADL's, Domain models, etc.).

Architectural Knowledge is mainly represented as documented knowledge in the form of an Architecture Description (IEEE/ANSI, 2000) or Architecture Documentation (Clements et al., 2002). An architecture document has several benefits for AK sharing as it allows for: (1) asynchronous communication (not face-to-face) among stakeholders to negotiate and reason about the architecture; (2) reducing the effect of AK vaporization (Jansen et al., 2008b); (3) steering and constraining the implementation; (4) shaping the organizational structure; (5) reuse of AK across organizations and projects; (6) supporting the training of new project members.

* Corresponding author.
E-mail addresses: anton@cs.rug.nl, gradius@fmf.nl (A. Jansen), paris@cs.rug.nl (P. Avgeriou), mail@jansalvador.nl (J.S. van der Ven).
URL: http://search.cs.rug.nl.
[1] Tel.: +31 50 363 3968; fax: +31 50 363 3800.

0164-1212/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2009.04.052

However, when systems grow in size and complexity, so does the architectural documentation. In such large and complex systems, this documentation often consists of multiple documents, each of considerable size, i.e. tens to hundreds of pages. Moreover, it becomes more complex, as within and between these documents, there are many concepts and relationships, multiple views, different levels of abstraction, and numerous consistency issues. Current software architecture documentation approaches cannot efficiently cope with this size and complexity; they are faced with a number of challenges that are outlined here and elaborated in Section 2:

(1) Creating understandable architecture documentation (Clements et al., 2002);
(2) Locating relevant Architectural Knowledge (de Boer and van Vliet, 2008);
(3) Achieving traceability between different entities (Hofmeister et al., 2005);
(4) Performing change impact analysis (Tang et al., 2005b);
(5) Assessing the maturity of the design (Bass et al., 2003);
(6) Trusting the credibility of the information (Lethbridge et al., 2003).

The research problem we address in this paper, is how to manage AK documentation of large and complex systems, in order to deal with these challenges. To partially tackle this problem, we propose an approach that enriches documentation with formal knowledge. The approach consists of a method supported by a tool suite. The key idea of this approach is to enrich software architecture documents by making the AK they contain explicit, i.e. capture this knowledge in a formal model. This formalized AK in turn is used to support the author and reader of the software architecture document with dealing with the aforementioned challenges. The proposed approach is complimentary to current architecture documentation approaches, as it builds upon them in order to transform documented into formal knowledge.

The usage of the process and the tool are demonstrated through a large and complex industrial example. We provide empirical evidence for the benefits of the approach through a quasi-controlled experiment in the context of this example. For reasons of scope and paper size, we only focus on one of the challenges (understandability).

The rest of this paper is organized as follows. Section 2 presents the aforementioned challenges of software architecture documentation in more detail. The next section introduces our method for enriching software architecture documentation with formal AK while Section 4 presents the accompanying tool, the Knowledge Architect. Section 5 explains how our approach, i.e. our method and tool addresses the aforementioned challenges. To exemplify the approach, Section 6 presents an example of the application of our method for a large, complex, and industrial system. We validate our approach with respect to one of the challenges using a quasi-controlled experiment in Section 7. In Section 8, related work is presented and the limitations of the approach are discussed in Section 9. The paper ends with directions for further work in Section 10.

2. Challenges for software architecture documentation

As described in the previous section, the research problem we deal with, is the inefficiency of current software architecture documentation approaches to deal with large and complex systems. We have broken down this problem into a set of challenges, which are elaborated in the following paragraphs:

- **Understandability.** Documentation always loses some of the intentions of the author when someone else reads it. As the size of documentation increases when systems become larger and more complex, the understandability of the documents becomes more challenging (Clements et al., 2002). Especially when stakeholders have different backgrounds, the language and concepts used to describe the architecture might not be understandable to everyone. Although good references and glossaries can help to improve the understandability, just reading the documentation often leads to ambiguities and differences in interpretation.
- **Locating relevant AK.** Finding relevant AK in (large) software architecture documentation is often problematic. The knowledge needed is often spread around multiple documents (de Boer and van Vliet, 2008). The first obstacle is to *find the relevant documents* in the big set of documents accompanying a system. The practice of informal sharing these documents through e-mails or shared directories complicates matters, leading to a situation where different people have different versions of the same document. The second obstacle is to *locate the relevant AK within these documents*. Although a clear documentation structure, glossary, and outline certainly helps, software architecture documents lack the required finer granularity for locating the *exact* AK.
- **Traceability.** Providing traceability between different sources of documentation is difficult (Hofmeister et al., 2005). In practice, the lack of traceability usually occurs between requirements and software architecture documents, since it is often unclear how these documents relate to each other. Text and tables have a limited ability to communicate different relationships. Figures (e.g. in the form of models or views (Clements et al., 2002)) inside architectural documentation are more effective in communicating relationships within or between documents. However, the semantics of these models and views are usually not explicit and therefore decrease the understandability.
- **Change impact analysis.** It is often necessary to predict the impact of a change on the whole system. Therefore, we need to analyze which parts of the architecture are influenced when an architectural decision is made or reconsidered (Tang et al., 2005a). Since documentation usually does not make these decisions and their relationships explicit, making a reliable change impact analysis is often very hard. The lack of traceability between the different architecture elements further exacerbates this problem.
- **Design maturity assessment.** Evaluating the maturity of an architecture design is difficult as there is no overview of the status of the architecture with respect to its conceptual integrity, correctness, completeness and buildability (van der Ven et al., 2006a; Bass et al., 2003). These types of qualities are different than run-time qualities (e.g. performance) or design-time qualities (e.g. modifiability) in that they are inherent to the architecture per se. Therefore they are quite complex qualities and usually difficult to assess through scenario-based evaluation methods (Bass et al., 2003). To make matters worse, the size and complexity of an architecture document directly influences these qualities and their assessment.
- **Trust.** Architectural documentation is constantly evolving and needs to be kept up to date with changes in the implementation and the requirements. In large and complex systems, changes occur quite often and the cost of updating the architecture document is sometimes prohibitive. Therefore, the document is quickly rendered outdated and the different stakeholders (e.g. developers and maintainers) lose their confidence in the credibility of the information in it (Lethbridge et al., 2003).

The challenges comprise the starting point for the remaining sections in this paper. An overview of the different sections and their relationships is illustrated in Fig. 1. On the top left, the challenges described in this section designate the problem statement. The next two sections describe our approach, consisting of a

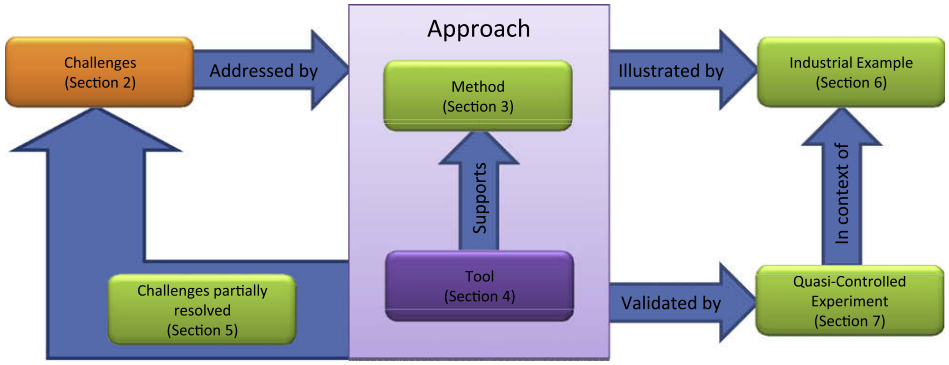

Fig. 1. Overview of the paper.

method (Section 3) and a tool (Section 4), for enriching documentation with formal AK. In Section 5, we describe how our approach partially resolves the six challenges. An industrial example presented in Section 6 helps to illustrate the approach while a (partial) validation through a quasi-controlled experiment is described in Section 7.

3. Enriching documentation with formal AK

A major cause of the inefficiency of current software architecture documentation approaches is the fact that they focus on documented and not formal knowledge. While documented knowledge can be managed by humans, this management does not really scale up when the size and complexity of the documentation increases. On the other hand, formal knowledge is more appropriate for automated processing and can handle scalability issues much more effectively. Consequently, formal knowledge in large and complex system can be automatically managed by appropriate tools that in turn support understanding AK, locating and tracing it, as well as analyzing and keep it up-to-date.

The key idea behind our approach is to add formal knowledge to existing documented knowledge in order to facilitate automated processing that scales efficiently and deals with the aforementioned challenges. Formal knowledge is added through annotating the existing documented AK sources according to a formal metamodel. This is different than creating formal AK from scratch, e.g. as done by Tyree and Akerman (2005), because we essentially reuse the existing AK and build formal AK upon it. Our approach is comprised of a method that describes the activities that need to be undertaken, accompanied with a tool that provides possibility to annotate documents. Next, the activities of our method are described.

(1) **Identify documentation issues.** The first activity in our method concerns identifying the problems in managing AK, starting from the six generic challenges presented in Section 2. Each one of these challenges can be refined into the specific problems the organization is facing. Not all six challenges must be necessarily dealt with; each organization can choose and emphasize on specific challenges. Furthermore the list of challenges discussed in this paper is not exhaustive; additional challenges can be considered according to the specific organizational context. After the challenges have been described in an organization-specific way, a number of use cases for managing AK need to be identified that will help to address the challenges. For example, we can derive specialized use cases on tracing particular types of organization-dependent AK such as risks and assumptions. As a starting point for selecting use cases, we propose our previous work on an abstract AK use case model that describes several possible uses of AK (van der Ven et al., 2006a). Since these use cases are rather abstract, they also need to be translated into the particular context of the system, by taking into account the sources that contain the AK.

(2) **Derive a domain model.** Based on the identified AK use cases, we derive a domain model consisting of concepts (i.e. Knowledge Entities (KE)) and their relationships that describe relevant AK. The domain model and the use case model are intertwined in the sense that the elements of the domain model should be used as specified in the identified AK use cases. Fig. 2 presents the basic model that can be used while constructing a specific domain model. This activity aims at producing a domain model (and thus the relevant AK) that is organization-dependent. This allows for the reuse of existing concepts and terminology within an organization across different projects. It allows an organization to use the domain model as a "standard" reference model to synchronize their terminology within the organization.

(3) **Capture AK.** Once a domain model is derived, AK can be captured that adheres to the domain model. It is very important to minimize the effort required to capture this knowledge. To achieve this, automation in the form of tool support plays a crucial role. Tools can substantially reduce the required effort by (semi-)automatically capturing AK. Typically, this involves information extraction techniques (e.g. (de Boer and van Vliet, 2008)) and assisting a user with producing AK (e.g. Tyree and Akerman (2005); Zimmermann et al. (2008)).

(4) **Use AK.** The goal of this activity is to *use* in practice the use-cases identified in activity 1 and thus deal with the corresponding challenges presented in Section 2. This activity involves consuming both documented and formal AK. The combination of these two types of knowledge should deliver more value as compared to the sole consumption of documented AK.

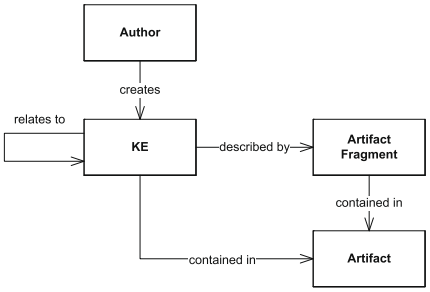

Fig. 2. The basic AK model.

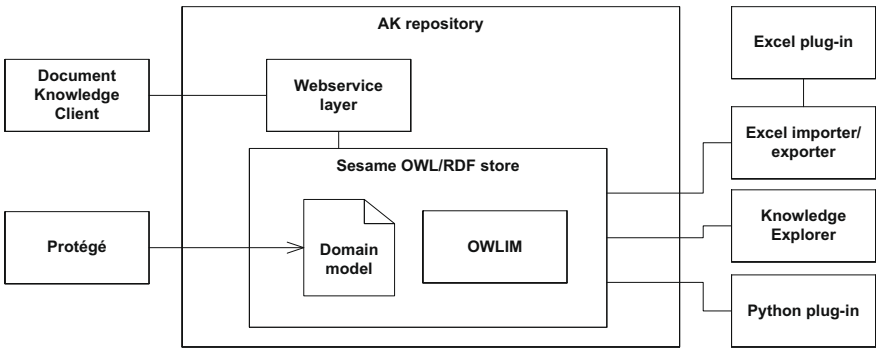

Fig. 3. The Knowledge Architect tool suite.

(5) **Integrate AK.** The domain model describes the relevant AK for a set of AK use cases. The different AK elements in the domain model are not always confined only to software architecture documents. Other sources may also contain valuable AK, e.g. analysis models, presentation slides, architectural models, wikis, discussion fora, and e-mails. Integrating the AK of software architecture documents with these sources enables a more complete representation of the knowledge.

(6) **Evolve AK.** A software architecture constantly evolves due to new developments and insights. Hence, there is the need to evolve the AK. This includes removing outdated knowledge (de Holan and Phillips, 2004) and updating relevant knowledge. So, both documented and formal AK should be kept up-to-date and in sync with each other. The challenge is to streamline this process to reduce the effort required. For example, in the context of architecture documents, this means finding a way to deal with cut and paste actions inside architecture documents and reflecting this in the related formal model.

The first four activities of the method (i.e. identify documentation issues, derive domain model, capture AK, and use AK) comprise the basic iteration where AK is produced and used. The final two activities comprise the next iteration where AK is integrated and evolved. In the remainder of this paper, we only discuss the first part (activities 1–4) and leave out the integration and evolution activities (i.e. 5 and 6). We have made this selection in order to scope the paper down to the basic iteration. By studying the first four activities, we can see whether the method works and brings the expected benefits. As further work, we plan to do additional research on the last two activities. Collecting a large amount of formal AK using the first four activities will provide us with a basis to experimenting with for the integration and evolution activities. The next section presents a tool suite that supports the outlined method.

4. The Knowledge Architect

4.1. Introduction

The Knowledge Architect is a tool suite that supports our proposed method by creating, using, and managing AK across documentation, source code and other representations of architectures. We briefly outline the tool suite and then explain how its different parts support the different activities of the method. The heart of the tool suite is an AK repository which provides various interfaces for tools to store and retrieve AK. The AK itself is represented in terms of fundamental units: the Knowledge Entities (KEs). Different tools can interact with the AK repository to manipulate the KEs:

- **The Document Knowledge Client** is a plug-in for Microsoft Word and enables the capture (by annotation) and use of AK within software architecture documents. The validation experiment in Section 7 focuses on the Document Knowledge Client.
- **The Analysis Model Knowledge Clients** supports capturing (by annotation) and using AK of quantitative analysis models. Specifically, there are two of such clients: a plug-in for Microsoft Excel (Jansen et al., 2008a) and a plug-in for Python.
- **The Knowledge Explorer** is a tool for analyzing the relationships between KEs. It provides various visualizations to inspect KEs and their relationships.

Fig. 3 presents an overview of how the various tools are related. The AK repository is the central point for storing and retrieving AK. It is built around Sesame, an open source RDF store (Broekstra et al., 2002). Sesame offers functionality to store and query information about ontologies (Antoniou and van Harmelen, 2004). Domain models are modeled as ontologies, which are expressed in OWL (Web Ontology Language) (Antoniou and van Harmelen, 2004). The Protégé tool is used to create the OWL definition of the domain model, which is subsequently uploaded to Sesame. To provide some intelligence in the AK repository, Sesame is extended with the inferencer OWLIM (Kiryakov et al., 2005), which offers OWL lite (W3C, 2004) reasoning facilities. The inferencer is mostly used to automatically generate the inverse relationships that exist between KEs. In this way, a user does not has to manually define them. The Document Knowledge Client uses a custom layer on top of Sesame to access the KEs. This layer provides a more high-level interface to Sesame; no tool developer is needed to understand the querying and low level storing mechanism of Sesame.

The Knowledge Architect tool suite can be used to support the activities of the proposed method, except for the first one. Activity 1 (Identifying documentation issues) is not supported, since it is a manual activity of refining the challenges and selecting the relevant use cases for AK. The AK repository is used to store the domain model resulting from activity 2.

The capturing of AK (activity 3) is supported by the Document Knowledge Client and Analysis Model Knowledge Clients that capture AK from Word documents, Excel analysis models and Python programs.

Using AK (activity 4) is supported by different parts of the Knowledge Architect, depending on the specific use cases that have been selected. For example, the Knowledge Explorer can be used to search for specific AK elements, while the Document Knowledge Client can be used to assess the completeness of the AK.

Integration of AK (activity 5) is naturally implemented in the Knowledge Architect through the central Knowledge Repository that collects all AK, and the combination of the various plug-ins, that store and retrieve the knowledge.

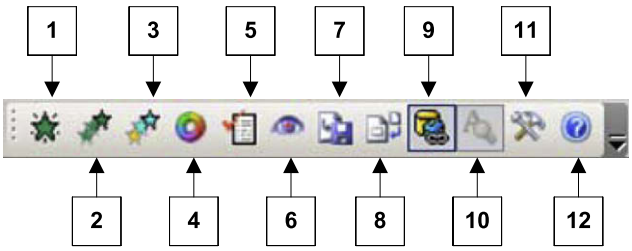

Fig. 4. The Knowledge Architect Word plug-in button bar.

Evolving AK (activity 6) is mostly supported by the Knowledge Explorer, which visualizes the interdependencies between the AK elements and thus facilitates change impact analysis. Changes to the AK can then be edited using the Document Knowledge Client and Analysis Model Knowledge Clients. The central Knowledge Repository is also useful for evolving the AK, allowing for easy management and identification of out-of-date AK and providing a history of its evolution.

4.2. Document Knowledge Client

The Document Knowledge Client[3] is a tool to capture and use explicit AK inside Microsoft Word 2003. The plug-in adds a custom button bar (see Fig. 4) and provides additional options in some of the context-aware pop-up menu's of Word. The tool automatically adapts at start-up time to the domain model used in the AK repository.

Fig. 4 presents the buttons that give access to the functionality of the Word plug-in. In short, they give access to the following functionality:

(1) Add current selected text and/or figure(s) as a new KE.
(2) Add current selected text and/or figure(s) to an existing KE.
(3) Create a KE table at the end of the document.
(4) Color the text of the KEs based on their type.
(5) Color the text of each KE based on its completeness.
(6) Show a list of KEs in the current document.
(7) Export KEs of the document to a XML file.
(8) Import KEs from the document into the connected AK repository.
(9) Connect to an AK repository.
(10) Read annotations from the current active document, i.e. enable the plug-in for the current active document.
(11) Open the settings menu.
(12) Display the plug-in version and authors information.

4.3. Knowledge Explorer

Typically, the size of an AK repository will be considerable containing thousands of KEs. Finding the right AK in such a big collection of KEs is not trivial. Hence, there is a need for a tool to assist in exploring an AK repository. The Knowledge Architect Explorer is such a tool. In this subsection, we briefly explain how this tool works and what kind of techniques are used to deal with the size of an AK repository.

Fig. 5 presents a screenshot of the Knowledge Explorer. On the left hand side the search functionality is shown. Users can use the see-as-you-type search box on the bottom left to look for specific KEs. The resulting KEs of this search action are shown in the list on the left hand side. The results can be filtered using the drop down box on the left, thereby reducing the size of the found results. The filtering is based on the type of the AK. The available options are presented based on the used domain model.

Double clicking on one of the search results focuses the visualization in the middle part of the figure on the selected KE. The selected KE (i.e. DD26) is indicated with a red background color. The middle visualization shows how the selected KE is related to other KEs. Double clicking on these related KEs changes the focus of the visualization accordingly.

The relationships that are shown depend on so-called "pillars". The pillars are the concepts of the domain model that are selected from a list on the top right and visualized as gray pillars in the middle. In the case of Fig. 5, these pillars are the Alternative, Decision Topic, and Requirements concepts. The pillar concept allows for easy inspection of whether a KE is (in)directly related to other KEs of a specific type. For example, this allows for checking whether a requirement eventually leads to a specification. This is simply achieved by only enabling the requirement and specification pillar. To get additional information about a KE, the mouse can be hovered over a KE and a pop-up window will present this information.

Another way to deal with the size of the AK repository is by using the list found in the middle right. This list presents all the KE authors and provides the opportunity to either include or exclude KE from specific authors for the visualization in the middle.

The last mechanism that helps dealing with the AK repository size is the slider on the middle right. This slider controls the distance at which a KE is no longer considered related to the selected KE. This distance is defined as the maximum number of relationships that may be followed to find a related KE. By moving the slider to the right, more distant related KEs are visualized, whereas moving the slider to the left reduces this number.

5. Challenges resolved

The method and the tool of the proposed approach aim at resolving the challenges presented in Section 2. In this section we outline how this takes place at a general level while in Section 6 we will go into the details of these challenges for a specific organization. Each challenge is addressed by the proposed method and tool as described below.

- **Understandability**
  - *Method.* The domain model derived in activity 2 of the method provides a common language for communication. This makes an architecture design easier to understand, as all concepts are defined in a clear way and are related to other concepts. The understandability is further increased when people become aware that they have to be strict when annotating their text (activity 3). This increases the clarity and unambiguity of the text. Also, when accessing and using the annotated documents (activity 4), the understandability is expected to increase, as described in the experiment in Section 7.
  - *Tool.* The Knowledge Explorer enhances understandability by visualizing the relationships between the different KE instances through the documentation. This offers the opportunity to gain insight into the architecture in a way that is hard to achieve by simply reading a software architecture document. The Document Knowledge Client improves understanding by offering traceability support, additional rationale, and meta-data about KE instances. In Section 7, we empirically validate whether this tool enhances the understandability.

[3] Downloadable from http://search.cs.rug.nl/griffin.

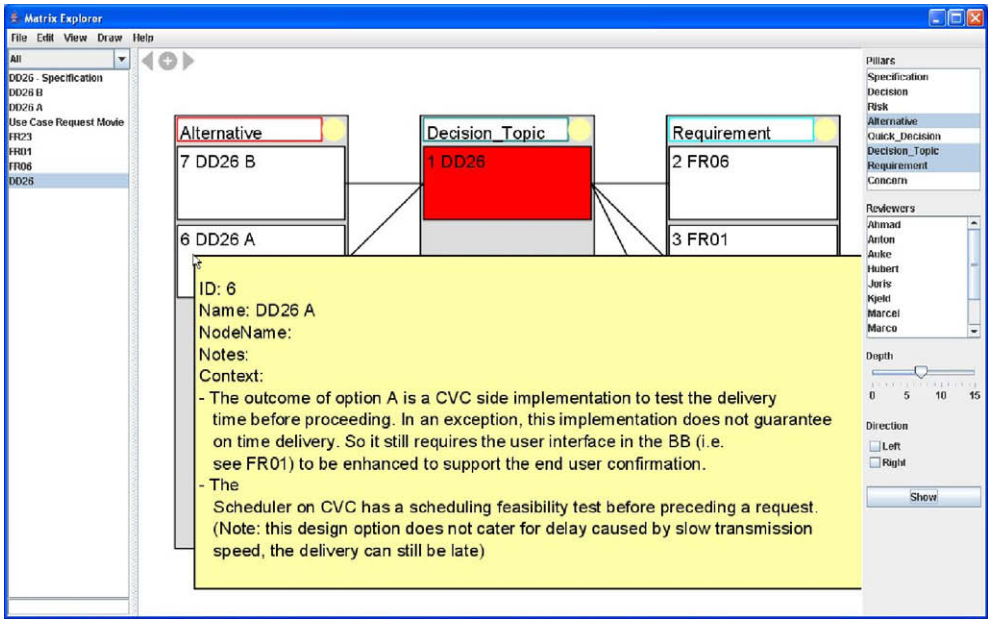

Fig. 5. The Knowledge Explorer.

- **Locating relevant AK**
  - *Method.* The method makes finding relevant AK easier due to the classification of the knowledge and the relationships the KEs have with each other. The classification allows one to scope the search for relevant AK to specific types of knowledge. This improves the quality of search results. In addition, the formal AK model allows to link search results to other related AK, thereby making it easier to find and understand the context of the knowledge.
  - *Tool.* The Knowledge Explorer offers search by keyword and by KE category (see Section 4.3), in order to find knowledge. Also, relevant AK can be found by following the relationships of KEs. The Document Knowledge Client can color KE instances making them easily findable on a document page (see Section 4.2). In addition, the tool can create a table with the different KE instances, using different orderings at the end of document. The KE instances in this table are provided with navigable links to their source, making the locating of relevant AK easier.
- **Traceability**
  - *Method.* The method does not only focus on capturing KE instances, but also on capturing the relationships among these instances. In doing so, the resulting AK model provides traceability among the AK (even through different sets of documentation, as described in activity 5).
  - *Tool.* The Document Knowledge Client supports people in creating (see Section 6.4) and using traceability information inside documents (see Section 6.5.1). Apart from the documents, the Knowledge Explorer tool supports analysis of traceability knowledge (see Section 4.3).
- **Change impact analysis**
  - *Method.* An important form of AK are architectural (design) decisions. Once these decisions are captured in a formal model (i.e. activity 3 of the method), assessing the impact of changing such a decision becomes easier. For example, techniques like Bayesian belief networks can then be employed to predict the impact of architectural design decisions (Tang et al., 2005b).
  - *Tool.* The impact of changes can be analyzed in the Knowledge Explorer (see Section 4.3). Selecting a changed KE instance (e.g. a requirement) in the tool will visualize the related (and potentially affected) knowledge (e.g. decisions).
- **Design maturity assessment**
  - *Method.* The method helps with assessing the maturity of a design. For completeness, automatic model checking can be used to asses what kind of AK is likely to be missing. To assist in assessing the correctness and consistency of the architecture design extensive formalization is required to model the semantics of the behavior of the designed system.
  - *Tool.* The Document Knowledge Client offers a completeness check, status field, and space for review comments to support such an assessment. We refer to Section 6.5.2 for an in-depth description of how the client supports this assessment.
- **Trust**
  - *Method.* The method helps with addressing the trust challenge by offering the possibility to attach meta-data to the captured and formalized AK. This facilitates the different stakeholders to investigate the author of the knowledge and the date it was created, and decide whether or not to trust it. Another example is aligning the process with KE instances by having a status field describing the status a KE has in this process. For example, Kruchten (2004) proposes to associate a status (e.g. Idea, Tentative, Decided, Approved, Challenged, Rejected, Obsolesced) to a decision.
  - *Tool.* The knowledge repository maintains a rich history of the KE instances, thus establishing how up to date they are. For example, the Document Knowledge Client can track the use, changes, and comments to individual KE instances, thereby providing a history that is suitable to judge the credibility of the knowledge. Also, by making it easier to assess the KE of an architecture through the explorer, it is easier to gain trust in the document at hand.

Fig. 6 presents a visual summary of the relationships between the challenges, activities, and tools. The challenges are depicted on the left side, the activities in the center, and the associated

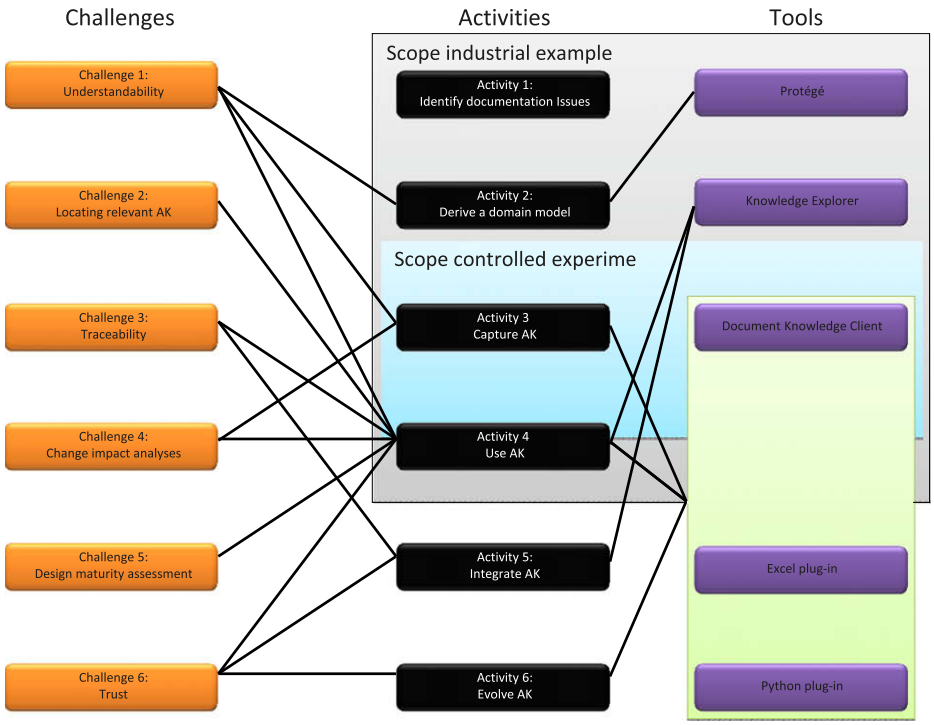

Fig. 6. Overview of the approach and its validation.

tools on the right side of the figure. Besides these relationships the figure also illustrates the scope of the upcoming two sections: the industrial example of Section 6 and the quasi-controlled experiment of Section 7.

6. The LOFAR example

In this section, we present an example of a real, large, and complex system. First, we present an introduction of this system. Then we present how activities 1–4 of our method (see Section 3) are applied in this context. We also outline, where appropriate, how the tooling of Section 4 is used to support the activities of our method.

6.1. Introduction

The industrial example investigated in this paper is LOFAR (LOw Frequency ARray) (Lofar website): a new radio telescope under construction by ASTRON, the Netherlands Institute for Radio Astronomy. LOFAR is rapidly becoming a European effort, with France, Germany, the United Kingdom, and Sweden having funded stations, with others to be added soon. What makes LOFAR interesting from a software architecture perspective is the fact that it is the first of a new generation of software telescopes (Butcher, 2004). Software is of paramount importance in the system design, as it is one of the crucial design factors for achieving the ability to communicate and process the 27 Tflop/s data stream in real-time to be fed into scientific applications. The architecture of this large and complex system is described in many different documents, ranging in scope from the entire system and particular subsystems to specific prototype analysis.

6.2. Activity 1: identify documentation issues

The first activity entails identifying the current issues with respect to using the architecture documentation. The challenges outlined in Section 2 are manifested in the LOFAR project as follows:

- **Understandability.** Creating a radio telescope that uses cutting edge technology involves many different specialists, each coming from a very different background, e.g. astronomers, high performance computing specialists, antenna specialists, industrial manufacturing experts, politicians, and embedded systems engineers. Hence, creating an understandable software architecture is vital for communicating and thereby creating consensus about the design among the stakeholders.
- **Locating relevant AK.** The architectural documentation of the LOFAR system consists of multiple documents, which in total encompasses over 1000 pages. Locating relevant AK is very hard simply due to its size.
- **Traceability.** The architecture description of LOFAR is split in separate documents for the top-level and individual sub-systems. Finding out what *exactly* the relationships are between these documents is very hard. It is especially difficult to understand how particular requirements are addressed in the architecture design.
- **Change impact analysis.** Predicting the impact of a design change is a major issue for LOFAR, as it forms a critical part of risk management. For example, a major risk is a change in the available budget, which has ramifications to the viability of the telescope design. Change impact analysis is needed to identify these ramifications.
- **Design maturity assessment.** At the time the investigation for this example took place, an important issue for ASTRON was to know whether the design was mature enough to be built or if additional design activities were needed.
- **Trust.** The design time for the LOFAR telescope is around 10 years, with an expected minimal operating time of 15–20 years. During this period, the telescope and its software will be constantly upgraded to improve performance. Hence, having

up-to-date, trustworthy AK will play a crucial role in the future of the telescope, as this partly defines the scientific relevance and success of the instrument.

After describing how the six challenges are manifested in the LOFAR project, we identified a number of use cases that help to address the challenges. We started from the use case list of van der Ven et al. (2006b) and derived a prioritized list of project-specific use cases. Based on this, we decided together with ASTRON to focus our effort on the use case: *Perform incremental architectural review*. ASTRON wants to perform better and more efficient architectural reviews. As stated in van der Ven et al. (2006b), this use case makes use of three other use cases: *Perform a review for a specific concern*, *View the change of the architectural decisions over time* and *Identify important architectural drivers*. This main use case touches upon three specific concerns (the aforementioned challenges): traceability, design maturity, and understandability.

Architectural reviews in ASTRON take place in two stages: first, the reviewers individually review one or more architectural documents and create comments about them; second, a review coordinator collects these comments and organizes a review meeting to discuss the most pressing issues. This use case focuses on supporting the first stage of the review process by enriching the used documentation using the Document Knowledge client (see Section 4.2). This helps the reviewers in a better and more efficient preparation for the review meeting. To increase efficiency, the document review can take advantage of tracking which KEs have been found consistent, complete, and correct, i.e. assessing the design maturity. The coloring of these KEs allows the reviewers to focus more easily on that part of the architecture description that requires further attention. Furthermore, providing traceability and easy spotting of relevant AK can improve the understanding a reader has of the software architecture.

6.3. Activity 2: derive a domain model

To discover what AK is relevant in the LOFAR system, we investigated the AK used and documented in the system taking into account the use cases from the previous activity. Independently from each other, two of the authors and a software architect of ASTRON examined a part of the architecture documentation. With a marker pencil, they annotated the text and/or figures that represented KEs. In the sideline of the document, they wrote down the name of the concept they believed this annotation to be an instance of. Prior to this, no deliberations were made on these concepts.

After completing this exercise, we compared the annotations and associated concepts with each other. The annotations made by the independent reviewers were surprisingly similar. Although the names of the concepts differed, the meaning of most of them were similar. Using affinity diagrams, we grouped the concepts. In case of doubts, the original pieces of text annotated were revisited and compared with each other. The aim of this exercise was to come up with the minimum set of concepts that was good enough to cover all the annotations.

The end result, i.e. the derived concepts and their relationships are presented in Fig. 7. Each concept inherits from a KE, as modelled in Fig. 2. Therefore the domain model for AK, specific to the LOFAR architecture documentation, is comprised of the following concepts:

- **Concern.** A concern is an interest to the system's development, the system operation, or any other aspect that is critical or otherwise important to one or more stakeholders.
- **Requirement.** A requirement is something that is explicitly demanded from the system by a stakeholder.

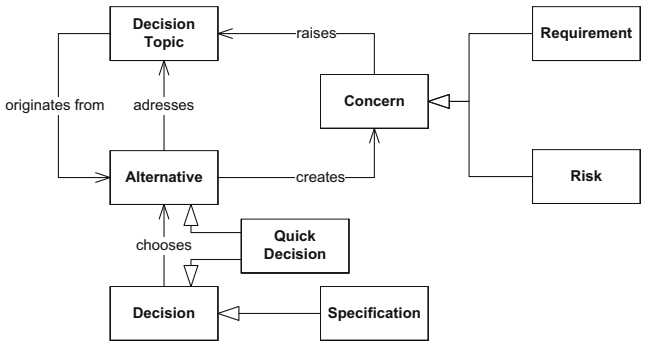

Fig. 7. A domain model for AK in documentation.

- **Risk.** A risk is a special type of concern, which expresses a potential hazard that the system has to deal with.
- **Decision topic.** A scoping of one or more concerns, such that a concrete problem is described. Often stated as a question, e.g. what is the contents of the data transport layer?
- **Alternative.** To solve the described problem (i.e. a decision topic), one or more potential alternatives can be thought up and proposed.
- **Decision.** For a decision topic there are sometimes multiple alternatives proposed, but only one of them can be chosen to address the described decision topic. The decision outlines the rationale for this choice.
- **Quick decision.** Often only one alternative is described to address a decision topic. Providing rationale for such an alternative is often lacking. The mere fact that the architect only describes a single alternative in the document, implicitly indicates that the architect has chosen this alternative as the one to use. Thus the alternative becomes a decision in its own right, i.e. a quick decision.
- **Specification.** A special kind of decision is a specification. It indicates the end of the refinement process for the software architecture. Any concerns coming up from the alternative chosen are in principle the responsibility of the detailed design.

6.4. Activity 3: capture AK

Capturing AK with the Document Knowledge Client involves the *Add KE* and *Add to existing KE* buttons, but can also be performed by selecting a piece of text and right clicking and choosing the appropriate option from the pop-up menu. When adding a new KE, a menu appears, which allows the user to provide the following additional information about a KE:

- **Name** that identifies the KE.
- **Type** of the KE, which is one of the concepts of the domain model being used. This can be selected through a pull-down menu.
- **Status** of the KE, which describes the level of validity of the KE, and is selected from the following options:
  - **To be reviewed** the KE needs to be reviewed by someone else then the creator of the knowledge.
  - **Reviewed** the KE has been reviewed, but no verdict has been reached yet.
  - **To be discussed** the KE is controversial and should be discussed.
  - **To be checked** additional analysis is still needed to support the validity of the KE.
  - **Validated** the KE can be regarded as stable and trustworthy.
  - **Obsoleted** the KE is no longer valid.

- **Connections.** The user can add and remove relationships to other KEs. Based on the earlier defined KE type and the domain model, the tool determines the type of relationships that might be available for new relationships to other KEs. Creating a relationship to a related KE is a four step process. To illustrate this process, we take as an example a new KE of the "Requirement" type. The first step is to choose the type of relationship. In our example, this could be either the "raises" or "created by" (the inverse of "creates") relationship, as defined in the domain model (see Fig. 7). The second step is to determine the scope in which the target KE of the relation can be found, which is either the current document, or the whole AK. Usually, a self-containing architecture document will have most of its relationships to KEs within the current document. The third step is to search for the KE, which is based on partial name matching. The (intermediate) results of the search are presented in a table like fashion, such that all details of the found KEs can be inspected. The fourth step is to select one or more of the search results and confirm the creation of a relationship. The inverse relationship(s) will be automatically created and maintained by the tool.
- **Notes**, which are additional textual information about the KE. Usually these contain pointers to more information or comments about the validity of the KE.
- **Creator** of the KE, which is automatically determined by the tool, based on the current configured Word user.

6.5. Activity 4: use AK

The enriched documentation can be used to execute the use cases identified in the first activity. In this section we focus on the use case of performing an incremental architectural review, as discussed Activity 1. We first describe how using AK during architectural reviews, helps to deal with traceability and understandability issues. Next, we describe how the design maturity can be assessed during such a review.

6.5.1. Traceability and understandability

A KE can be edited or removed by choosing the appropriate option from the pop-up menu when right-clicking on the text of the KE. In the same menu, the relationships among KEs can be followed. Thus useful *traceability* among KEs is provided. Fig. 8 exemplifies this: under the "Connections..." the pop-up menu lists the relationships that a KE has, while clicking on them moves the cursor to the appropriate piece of text. This allows for a hyper-link style of navigation inside an architecture document. Navigating back to the originating KE is easy due to the automatically created inverse relationships.

To enhance the understandability of the document, the tool facilitates the recognition of existing KEs by coloring the text based on the KE type (button 4 in Fig. 4). Fig. 8 gives an example of the effect of this coloring. The colors used for each type can be configured in each AK repository. This improves understanding in two ways. Firstly, by simply browsing through an annotated document gives the reader a global understanding of where most relevant AK resides in the document. Secondly, by making the KEs and their type easy to spot, a reader (e.g. a reviewer) can straightforwardly guess the message, that the architect tries to communicate.

6.5.2. Design maturity assessment

The Document Knowledge Client can support the architect in assessing the completeness of the architecture description. Based on the domain model, the tool performs model checks to identify incomplete parts. For each KE inside the document, a completeness level is determined. The completeness levels are named after the colors that are used to color the text of the KE. To find out why the tool deems a certain KE to be incomplete, the user can inspect the "Completeness..." option of the context pop-up menu to see which rules are not adhered to. Fig. 9 presents an example of this. The tool distinguishes the following four completeness levels (ordered from high to low severity):

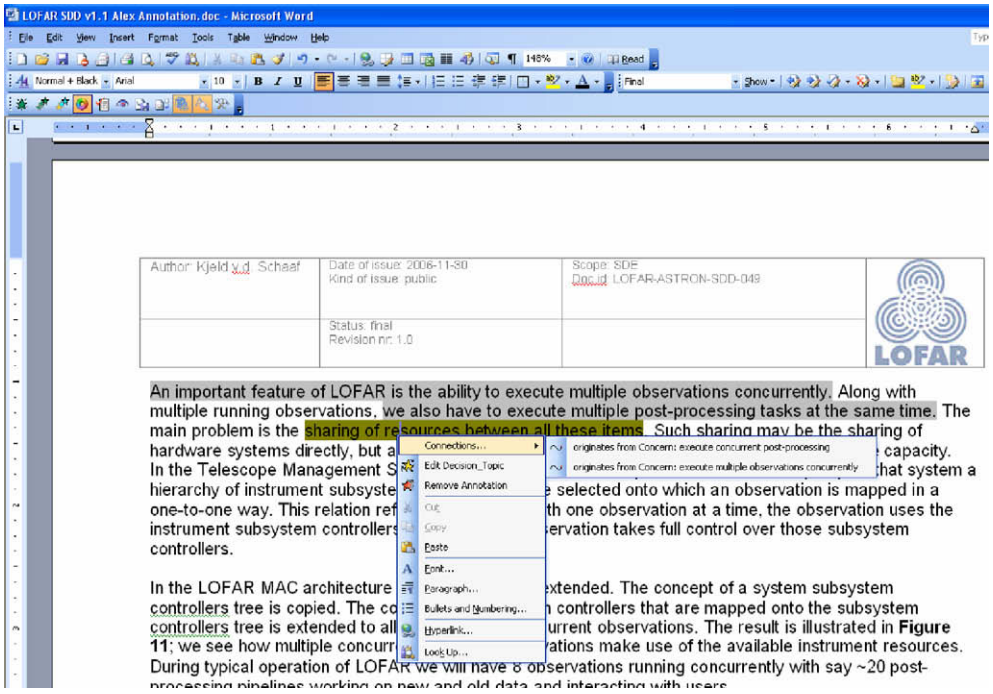

Fig. 8. A software architecture document with colored KEs and pop-up menu for tracing the relationships of a KE.

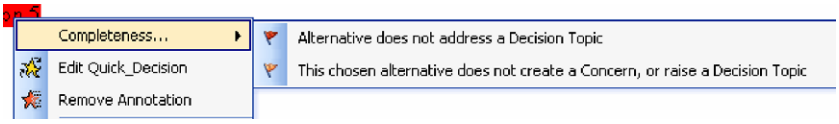

Fig. 9. Incompleteness information of a KE.

- **Red.** One or more primary rules are violated.
- **Orange.** The primary rules are adhered to, but one or more secondary rules are violated.
- **Yellow.** Both primary and secondary rules are adhered to. However, the KE has not achieved the status of "validated" yet.
- **Green.** Both primary and secondary rules are adhered to. In addition, the KE has been validated by a reviewer.

The distinction between primary and secondary rules is a pragmatic one. Primary rules are those that check whether the document is complete enough to provide a minimum level of traceability. This minimum level should ensure the existence of at least one reasoning path a reader could follow. Secondary rules focus more on the completeness of the architecture design. Both the primary and secondary rules depend on the specific domain model used, as they use the concepts and relationships of the domain model to detect missing information. The rules are evaluated inside the AK repository, which offers an infrastructure to easily add or remove new rules during run-time. For the ASTRON LOFAR Domain model (see Fig. 7), the following primary rules are used:

- All Alternatives address one or more Decision Topics each.
- All Decision Topics are addressed by at least one Alternative.
- All Decisions choose exactly one Alternative. This rule is not applied for a Quick Decision.
- All Decision Topics have an originating Concern or Alternative.

The following secondary rules are used:

- A Concern raises at least one Decision Topic.
- Concerns, that are not Requirements or Risks, are created by Alternatives.
- Chosen Alternatives and Quick Decisions that are not Specifications, either create at least one Concern,or raise at least one Decision Topic.
- Quick Decisions should not have "chooses" or "chosen by" relations to other KEs.
- A Quick decision should be the only Alternative addressing a Decision Topic.
- Exactly one Alternative should be chosen for a Decision Topic.

7. Quasi-controlled experiment

This section presents a quasi-controlled experiment to empirically validate a part of the presented approach. This experiment is conducted as an observational study. This section follows the controlled experiment reporting guidelines of Jedlitschka and Pfahl (2005). Since the experiment is only part of this paper, some parts of the reporting guidelines are already covered in other sections. In specific, the content of the structured abstract is part of the introduction, related work is discussed in Section 8, and future work is presented in Section 10.

7.1. Motivation

To validate our approach, we conducted a quasi-controlled experiment. The experiment focused on one of the identified challenges (understandability, see Section 2) and on a specific use case (performing incremental architecture review, see Section 6.1). In addition, the focus was on the Document Knowledge Client and did not involve the Explorer.

7.1.1. Problem statement and research objectives

The research question we answer with the quasi-controlled experiment is the following:

> *Does our approach for enriching software architecture documentation with formal AK improve the understanding of a software architecture description?*

We present the research objective using the template suggested in Jedlitschka and Pfahl (2005):

> **Analyze** the presented approach
> **for the purpose of** improving
> **with respect to** software architecture understanding
> **from the point of view of** the researcher **in the context** of the LOFAR example presented in Section 6.

7.1.2. Context

The context of the quasi-controlled experiment is the LOFAR system, as described in the previous section.

7.2. Experimental design

7.2.1. Goals, hypotheses, and parameters

In our experiment, we compare the understanding one has of the architecture when using a normal documentation approach as opposed to a documentation approach which includes the possibility for enriching the documentation. For this, we need a way to quantify the understanding (and associated communication) someone has of a software architecture. Achieving such a measurement for such a complex topic as a software architecture is very difficult. One activity in which the understanding of a software architecture plays a key role is that of an architectural review. Understanding the architecture is crucial for a reviewer's ability to judge an architecture. Hence, we can indirectly measure the understanding someone has of a software architecture by looking at how well he or she performs an architecture review.

Based on this assumption about the relationship between understandability and architectural review, we have formulated the following null hypotheses:

- H_{01}: Consuming formal AK makes an architecture review less efficient, i.e. #comments(ConsFormAK) < #comments(ConsDoc).
- H_{02}: Consuming and producing formal AK makes an architecture review less efficient, i.e. #comments(ConsProdFormAK) < #comments(ConsDoc).
- H_{03}: Consuming formal AK degrades the quality of a review, i.e. qualityComments(ConsFormAK) < qualityComments(ConsDoc).
- H_{04}: Consuming and producing formal AK degrades the quality of a review, i.e. qualityComments(ConsProdFormAK) < qualityComments(ConsDoc).

The associated alternative hypotheses are:

- H_1: Consuming formal AK makes an architecture review more efficient, i.e. #comments(ConsFormAK) > #comments(ConsDoc).
- H_2: Consuming and producing formal AK makes an architecture review more efficient, i.e. #comments(ConsProdFormAK) > #comments(ConsDoc).
- H_3: Consuming formal AK improves the quality of a review, i.e. qualityComments(ConsFormAK) > qualityComments(ConsDoc).
- H_4: Consuming and producing formal AK improves the quality of a review, i.e. qualityComments(ConsProdFormAK) > qualityComments(ConsDoc).

The experiment is embedded into ASTRON's normal development process and followed their normal procedures for an architectural review. This means that one person is the coordinator for the review. He or she receives the software architecture document from the architect and sends them out to the reviewers. The reviewers read the software architecture document and send their comments before a deadline to the coordinator. After all reviewers have sent in their comments, the coordinator makes a selection of these comments and arranges a meeting with the architect and reviewers to discuss the selected comments.

7.2.2. Independent variables

The experiment consists of two independent variables: (1) the (none) use of the tool (2) the (none) production of formal knowledge. We call the combination of these two variables a *situation*, i.e. a treatment in empirical research. To determine the effectiveness of our approach, we examine the following three situations:

- **Situation 1: consume documented/formal AK.** In this situation, the subjects use the Knowledge Architect Document Knowledge Client with an annotated version of the software architecture document. They are not allowed to create new annotations. Hence, the subjects only *consume* formal AK (Lago and Avgeriou, 2006).
- **Situation 2: consume documented AK and produce formal AK.** In this situation, the subjects use the Document Knowledge Client on a unannotated version of the document. They are encouraged to make their own annotations alongside their review. Hence, the subjects *produce* formal AK and only *consume* their own produced formal AK and the documented knowledge from the document.
- **Situation 3: only consume documented AK.** The subjects do not use the Document Knowledge Client and do not consume formal AK. They merely review the document, as in a "normal" review, but still read the document from a computer screen.

Situation 3 acts as a baseline to compare the performance of situations 1 and 2.

7.2.3. Dependant variables

The experiment uses two dependant variables for measuring the understanding of the architecture. They are based on the review comments of the subjects. First, we measure the *broadness* of this understanding by looking at the *quantity* of comments each subject makes in a limited amount of time, i.e. 1 h. Second, we measure the *deepness* of this understanding by rating the *quality* of the comments. This latter quality is defined as the extend a comment helps to improve the architecture and its description. The comments are rated by two people: the architect and a very experienced architecture reviewer. They give each comment a rating on a scale of 1–5, with 1 being the lowest quality rating and 5 the highest.

7.2.4. Experiment design

Each of the subjects perform two reviews. For this, we have split the software architecture document in two equally sized parts, i.e. Chapter 1 and Chapters 2 and 3, that describe different aspects of the architecture independent from each other. Each subject therefore performs two reviews, one for Chapter 1 and one for Chapters 2 and 3. Consequently, each subject only participate in a maximum of 2 out of 3 situations.

We designed the experiment in such a way that the subjects were evenly distributed over the 3 situations per document part. Table 1 presents this distribution. The experiment design is a semi-randomized design, as we put additional constraints on allowed assignments of subjects to situations. That is, each subject was randomly assigned to two *different* situations. The top and middle of Table A.1 in the appendix presents the resulting assignments of subjects to situations using a sort card randomization.

7.2.5. Subjects

In total 16 persons participated in the experiment. The subjects had different backgrounds: senior software engineers (subjects 8 and 15) and software engineers working on the LOFAR system (subjects 3 and 7), master students in software engineering who have participated in a course on software architecture (subjects 1, 2, 4, 6, 9, 10, and 11), and academic researchers of AK (subjects 5, 12, 13, 14, and 16). Hence, 4 of the subjects are practioners and the other 12 are academics. One of the master students (subject 6) knows the Document Knowledge Client, for he has been involved in its development. All other subjects were not knowledgeable about the tool.

7.2.6. Objects

The document used for the experiment was a recently created software architecture document of the LOFAR system (see Section 6.1). This document is not part of the set of documents we investigated for the domain model (see Section 6.3). Each subject was provided a laptop on which the Document Knowledge Client and the document was available. For subjects that were performing their review in situation 1 or 2, a one page supporting leaflet was provided to them. The leaflet explained the LOFAR domain model (see Section 6.3) and a very short manual on the workings of the Document Knowledge Client.

7.2.7. Instrumentation

Before the actual review started, a 40 min presentation explaining the experiment was given to the subjects. This presentation included an explanation of the Document Knowledge Client and the LOFAR domain model. Following was a small training exercise lasting for 20 min. In this exercise, the subjects used the Document Knowledge Client to annotate and use formal AK in a sample document.

To guide the subjects with capturing comments during their review a template was provided to them. The template was a simple table, with one row per comment. The reviewers were asked to fill the following two significant columns: comment text and comment type. The first column contains the text of the comment. In the second column, the reviewer classified his/her comment as

Table 1
Experimental design: #subjects per situation.

	Situation		
Chapter	1	2	3
1	6	5	5
2 and 3	5	6	5
Total	11	11	10

either a positive remark or as an improvement to the architecture (description).

To gather important qualitative information from the subjects, the experiment ends with a group discussion. We used the following checklist to ensure the discussion covered vital parts we were interested in:

- **General remarks** about the quasi-controlled experiment.
- **Bugs** the subjects encountered while using Document Knowledge Client.
- **Improvements** that could be made to our tool and approach.
- **Domain model** how good or bad it was for the specific review task.
- **Creating annotations** besides the provided ones, a situation not covered in our quasi-controlled experiment.
- **Future use**, i.e. whether the subjects would like to use the Document Knowledge Client again the next time when they are performing an architectural review.

7.2.8. Data collection procedure

The experiment was performed three times at separate days. The experiment started with the aforementioned 40 min presentation and a 20 min training exercise. After this first hour, the subjects started with their reviews in their assigned situation. Once the two review sessions were completed, the experiment was concluded with a 15–30 min wrap-up discussion session to collect the experiences of the subjects. All in all, including breaks, the entire experiment took 5–7 h per person.

For each review, the subjects had two hours of time. The review comments were collected at the end of the first and second hour. Due to time constraints of the ASTRON engineers, we limited their review time to a single hour. This is why in the rest of this experiment the focus is on this first hour.

7.2.9. Analysis procedure

In this experiment, we focus on the result of the review that was send to the coordinator; a list of comments and remarks about the software architecture document. By judging these comments, we quantify how well a reviewer had performed the review, and thus indirectly measure how well they understood the architecture.

We simplified the experiment analysis by making several important assumptions. Without these assumptions, we should use a non-parametric statistical test. However, seeing that we have very limited number of subjects, achieving significant results is most likely impossible. Hence, we want to use a parametric test. However, for this to work we need to make three assumptions. Firstly, as the metric for the quality of a comment the average score of both raters is used. Secondly, we assume that the number of comments per subject is an independent variable, i.e. the number of comments made by one reviewer does not influence the number of comments made by another reviewer. Thirdly, the number of comments for a situation has a normal distribution. Based on these three assumptions, we can use the student t-test (Sheskin, 2003) to statically test whether the encountered differences are significant. We use the one-tailed variant of this test, as we want to measure whether the found differences are statistically significant.

The student t-test calculates the chance that a similar result will be found when the experiment is repeated. In this paper, we call this chance the confidence level, which is defined as $1 - p$ with p being the so-called p-value. Most empirical researchers use a confidence threshold of 0.95 (i.e. 95% or $\alpha = 0.05$) as the minimum level to accept a hypothesis. For this paper, we use an α value of 0.05 to statistically accept a hypothesis, i.e. $p < \alpha$. For results with confidence levels between 0.80 and 0.95, we regard the results to be strong indicators for their associated hypothesis.

7.2.10. Validity evaluation

We improve the reliability and validity of the data collection in various ways. Firstly, we enabled the automatic file saving feature of Word on a short interval of 5 minutes to prevent losing either review comments or annotations due to crashes. Secondly, we ensured that assistance was available for the reviewers in the case they were confronted with problems, e.g. with understanding the working of the tool.

7.3. Execution

7.3.1. Sample

Table A.1 (see appendix) presents the raw data resulting from first hour of the experiment. The table presents a number of things for the two reviews (i.e. Chapter 1 and Chapter 2 and 3). First, it displays in which situation a subject was performing a review. Second, it presents the results in this situation after 1 h. Third, it shows for every subject the number of comments that have received a certain rating. The left number is the rating by the architect and the right number is the rating of the review expert. In total, 203 comments were collected for the first hour and 94 more in the second hour.

7.3.2. Preparation

The preparation went smoothly and followed the description outlined in the experimental design (see Section 7.2).

7.3.3. Data collection performed

The data collection performed followed the description of Section 7.2.8. There was one exception, subject 16 only performed the first hour of the review of Chapter 1, whereas 2 hours were planned. Since the analysis of this experiment concentrates on the first hour, this deviation has no influence on the experiment results.

7.3.4. Validity procedure

No crashes occurred during the experiment. Assistance with the Document Knowledge Client was needed during the first execution of the experiment, as the color scheme used to color KEs according to their type in the document was not clear. The supplied one page leaflet was updated to include this information.

7.4. Analysis: quantity

7.4.1. Descriptive statistics

Based on the results presented in Table A.1, we evaluate the quality and quantity of the comments. For the quantity, we count the number of comments per reviewer per situation. This number in turn is averaged over all the reviewers in a particular situation. Fig. 10 presents the resulting average number of comments of the reviewers per situation.

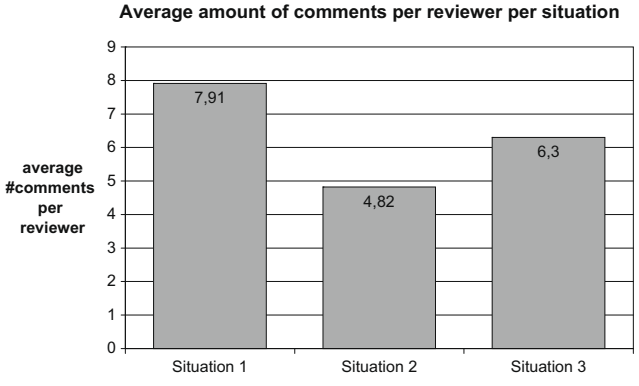

Fig. 10. Average number of comments of the reviewers per situation.

7.4.2. Data set reduction

The comments displayed in A.1 are those comments the reviewers themselves labelled as improvements and not as positive remarks. Since the positive remarks do not add value to the review process, we have left these out.

7.4.3. Hypothesis testing

Fig. 10 shows that, in the experiment, on average the subjects make more comments when consuming formal and documented knowledge (situation 1) compared to a normal review (situation 3). This supports hypothesis H_1. In addition, the subjects seem to make less comments when producing formal AK (situation 2) compared to a normal review in which only documented AK is consumed (situation 3). Hence, we reject hypothesis H_2 and consider the associated null hypothesis H_{02}. However, the question is whether these found differences are statistically significant.

To calculate the *t*-test value, the standard deviations of the results per situation are needed. In short order, these are: 7.66 (situation 1), 4.71 (situation 2), and 6.15 (situation 3). Based on these values , the data of Table A.1, and Fig. 10, we find the following confidence levels for hypotheses H_1 and H_{02}:

Hypothesis	Situations	Confidence	*p*
H_1	$1 > 3$	0.6980	0.3020
H_{02}	$2 < 3$	0.7296	0.2704

7.5. Analysis: quality

7.5.1. Descriptive statistics

We analyze the results for the *quality* of the comments by calculating an average comment rating score for each subject in each situation. Since The Pearson Product Moment Correlation Coefficient (Rodgers and Nicewander, 1988) between the two raters is rather low, i.e. $r = 0.29$. Hence, using the average is a rather conservative way to deal with this. We calculate this average using the following equation:

$$\frac{\sum_{i=0}^{i=n} cwa_i + \sum_{j=0}^{j=n} cwer_j}{2 * n}.$$

In this equation, n is the number of comments of a subject. cwa_i and $cwer_j$ are the ratings the architect and the review expert have given as quality of a comment, i.e. the values from Table A.1. Thus we calculate an average comment rating for each subject in each situation. In turn, these averages are used to calculate the average quality of a comment per situation. The results of this calculation are presented in Fig. 11.

7.5.2. Data set reduction

Note, that for subject 7 these numbers don't add up, as one comment was not rated by the architect. Since the average of both reviewers is used as the metric of the quality of a comment, we use for this comment only the quality rating given by the review expert.

Another complication in the quality calculation are subjects that have no comments, i.e. subject 9 for Chapter 1 and subject 8 for Chapters 2 and 3. For these two subjects an average quality of their comments cannot be determined. Hence, we exclude them from the calculation of the average comment quality per situation.

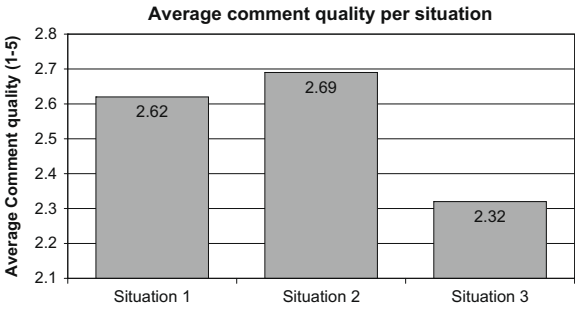

Fig. 11. Average quality of comments of the reviewers per situation.

7.5.3. Hypothesis testing

Comparing Fig. 11 with the quantitative results presented in Fig. 10 it is surprising that the average quality of the comments, although there are less, in situation 2 is higher than that in situation 3. This indicates that producing AK deepens the understanding of an architecture document, but reduces the broadness of this understanding.

To determine whether the found differences are statistically significant, we use the same student *t*-test as for the quantitative part. The standard deviations for the three situations are: 0.39 (situation 1), 0.62 (situation 2), and 0.28 (situation 3). Based on these numbers, we find the following confidence levels for hypotheses H_3 and H_4:

Hypothesis	Situations	Confidence	*p*
H_3	$1 > 3$	**0.9584**	**0.0416**
H_4	$2 > 3$	0.9292	0.0708

7.6. Interpretation

7.6.1. Evaluation of results and implications

For the quantity of the comments, we have hypotheses H_1 and H_{02}. Based on the results, we cannot statistically accept hypotheses H_1 and H_{02} However, the result does give a weak indication that an improvement in the number of comments for situation 1 over situation 3 is likely and the opposite is the case for situation 2 compared to situation 3. For completeness, we also calculated whether situation 1 is an improvement over situation 2. The confidence we find for this improvement is 0.8661, which is not statistically significant, but still a strong indicator that this difference might exist.

Based on the results for the quality of the comments, we conclude that there is a strong indication for H_4. However, the hypothesis lacks the confidence to be statistically accepted. For H_3 this is different as for this hypothesis the data does statistically support the hypothesis. Thus, the quality of the comments, on average, is better when consuming formal and documented AK than when only consuming documented AK. Consequently, we conclude that the understanding of the software architecture is deeper when using formal and documented AK than using only documented AK.

7.6.2. Limitations of the study: threats to validity

There are several threats to the validity of the quasi-controlled experiment. The following list presents these threats and our mitigation strategy to deal with them. We have categorized the threads as being either for internal or external validity.

Internal:

- **Learning effect.** The presentation of the LOFAR domain model and the exercise with the Document Knowledge Client before the start of the reviews might influence the behavior of the subjects in situation 3, since the subjects learn spotting relevant KEs. In addition, the use of the tool in situation 1 and 2 for the first review might influence a subject's second review perfor-

mance for the same reason. Since we do not want to complicate the execution of the experiment, we did not explicitly mitigate this issue.

- **Subjectivity of comment quality ratings.** The rating of a comment is a subjective measurement. This threat is mitigated by using the average rating of two persons instead of one to rate the quality of the comments.
- **A comment is not an useful unit.** A reviewer can comment on different parts of the architecture in a single comment. Hence, a comment might not be a useful measurement unit, as the same comment could be represented by multiple other comments. We manually inspect all the comments to detect and repair such cases by splitting them up in separate comments to mitigate this threat.
- **Personal bias of subjects.** One subject might be a much better reviewer than another subject. Hence, there is a personal bias of the subjects that might influence the experiment. To mitigate this threat, we let each subject perform his/her two reviews in different situations. Consequently, the personal bias is (partially) mitigated, as 2 out of 3 situations are influenced by this bias. For example, subjects 8 and 9 are excluded in the analysis of the quality, as they have only for one situation more than zero comments. If these subjects were included in the analysis then H_3 would have a confidence of 0.74 and H_4 0.55, since one contributes with a very high (3.75) score for situation 3 and the other with a very low (1) score for situation 1. Hence, including them contributes a big bias in our experiment, especially as they do not "compensate" in another situation.
- **Background bias of subjects.** Besides the aforementioned personal bias, the subjects are also biased by their background, i.e. their education and experience. We mitigate this effect by evenly distributing the subjects over the situations based on their backgrounds.
- **Bias of quality ratings.** The people rating the comments may favor one situation over another. We mitigate this effect by performing these ratings "blindly", i.e. the person rating does not know from which situation they come. In addition, these people did not participate as subjects in the experiment nor are they the authors of this publication.

External:

- **Domain dependency.** The experiment takes place in the context of radio telescopes. Hence, the findings might not be representative for other application domains.We try to mitigate this threat by having non-domain experts (i.e. master students) in our subject population. However, replication of the experiment in another application domain is advisable.
- **Scalability of the findings.** The quasi-controlled experiment is centered around one relative small (±50 pages) document. Hence, it is the question as whether these findings still hold for the entire architectural documentation. We have tried to partial address this by using a document that provides an overview of the entire software architecture and relates to more specialized documents.

7.6.3. Inferences

Besides the four hypotheses, there are some additional observations concerning the quasi-controlled experiment. Looking at Table A.1 and recalling the standard deviations presented earlier for the different situations, we see big differences in the performance of our subjects. For example, subject 9 produces 0 comments in a hour about Chapter 1, whereas subject 16 manages to produce 20 in the same amount of time and situation. Hence, the primary factor influencing the experiment seems to be a subject's reviewing capabilities. The situation in which this review takes place is more a secondary influence.

7.7. Lessons learned and discussion

To collect the non-quantified experiences of the subjects, the experiment ended with a discussion session (see Section 7.2.7). In this subsection, we present these experiences.

The subjects have two remarks regarding the setup of the experiment. The first remark is that we test the reviewers performance in situation 3, i.e. only consume documented AK, with the software architecture document displayed on a computer screen. Many subjects prefer to review a document on paper as this offers reading benefits computer screens cannot deliver (O'Hara and Sellen, 1997). In the design of the experiment, we use computer screen reading for all situations as not to influence the result by the distinction of reading on paper versus reading from a computer screen.

Secondly, some of the subjects felt restrained in situation 1, i.e. consume documented and formal AK, as they are not allowed to make annotations themselves besides the provided ones. We decided beforehand to leave out this case, as to have a clear distinction between consuming and producing formal AK.

Based on some of the subjects comments, we have improved the usability of the Document Knowledge Client. In short, these comments have lead to four significant changes. First, changing the type of a KE has become possible and no longer requires removal of the old KE. Second, a huge performance improvement has been made when switching between different documents. Third, the user interface menu for the connections is revised to make it easier to see and edit the connection a KE has. Fourth, the tool now supports some basic keyboard shortcuts.

The subjects indicate that in situation 2, i.e. produce formal AK and consuming documented and own produced formal AK, they operate in two different processes. In the first process, they read the document and try to make annotations of the AK they discover, i.e. they produce AK. In the second process, they read these annotations and the accompanying texts to distill review comments, i.e. mostly consume AK. Most subjects start of with the first process followed by the second process. However, in following iterations many subjects say to skip the first process altogether, as they feel slowed down by making annotations.

We asked the subjects on their opinion about the domain model. Generally speaking, the subjects were satisfied with the model. However, the software architecture document was lacking Decision Topics. Only with hard thinking these could be constructed. A problem is where to attach to the annotations of these non-existing Decision Topics.

Two of the subjects indicate that they find it hard to get an overview of the software architecture with the Document Knowledge Client. The explorer tool (see Section 4.3) might provide this wanted overview. However, this tool was not part of the experiment. Testing the suitability of the explorer for this role is future work.

We also asked the subjects if they would use the tool again after the experiment. Several subjects indicate that they would like to use the tool when writing a software architecture document as to improve the quality of it. The majority of the subjects would use the tool again for a review if the annotations would be provided. The remaining minority prefers to use paper for their reviews.

8. Related work

The approach presented in this paper is based on annotating documents to make their knowledge explicit. Similar approaches exists in the context of the semantic web in the form of annotation tools, e.g. MnM (Vargas-Vera et al., 2002) and Annotea (Kahan and

Koivunen, 2001). However, all of them focus on annotating web pages and plain text, not the Word documents in which software architectures are typically written.

Closely related to annotating is the field of Information Extraction (IE) (Cowie and Lehnert, 1996), where the challenge is to automatically annotate or extract information from documents or find relevant relationships between annotations and/or documents. Usually, this involves machine learning, natural language processing, and/or statistical techniques. For example, Ont-O-Mat (Handschuh et al., 2002) uses machine learning to semi-automatically annotate similar documents. Another example is the work of de Boer and van Vliet (2008), who use latent semantic analysis (Letsche and Berry, 1997) on (architectural) documents to find relationships between these documents. Based on these relationships an order of reading the documentation is suggested.

Most software architecture documentation approaches (Kruchten, 1995; Clements et al., 2002; Hofmeister et al., 2000) use architectural views to describe different aspects of an architecture. This is reflected in the IEEE 1471 standard, i.e. a recommended practice for architecture description of software-intensive systems (IEEE/ANSI, 2000). The approach presented in this paper can be used in conjunction with such approaches. Documentation approaches define for each view what concerns are of interest and how they should be described. Our approach provides the necessary glue in the form of traceability to relate the views and their underlying decisions (Duenas and Capilla, 2005; Jansen and Bosch, 2005) together.

Another form to support AK capturing in architecture documentation is by the use of templates. Tyree and Akerman (2005) present such a template for architectural decisions. However, templates typically have difficulty in making multiple relationships between different elements clearly visible. Visualizations like the one in the Explorer (see Section 4.3) are much more capable of dealing with multiple relationships.

In the last couple of years, several AK management tools have been developed. This includes web-based tools like PAKME (Babar et al., 2006) and ADDSS (Capilla et al., 2006), which focus on design patterns and architectural decisions, or more implementation focused tools like Archium (Jansen and Bosch, 2005) and AREL (Tang et al., 2007). The main difference between the Knowledge Architect and these AK management tools is that the Knowledge Architect uses specialized plug-ins to integrate with different AK sources, something these other tools do not do.

Compared to the meta-models of these AK management models or general meta-models like Kruchten's ontology Kruchten et al. (2006), the template from Tyree and Akerman (2005), or the CORE model (de Boer et al., 2007), the domain model used in this paper is rather simplistic. Both in the number of concepts and relationships and the provided detail. This is due to two reasons. Firstly, our domain model focuses on AK in software architecture documents in a specific organization instead of AK in general. Secondly, we strived for a minimal model that was just good enough as to make it easier to understand.

The just-in-time AK Sharing portal of Farenhorst et al. (2008) is similar to our AK repository; a central storing location for AK. The main difference between the two approaches is the assumed knowledge management strategy (Hansen et al., 1999; Babar et al., 2007). The just-in-time AK sharing portal focuses on a personalization strategy, whereas the Knowledge Architect is a codification strategy. Hence, the just-in-time AK sharing portal focuses on knowledge where the AK can be found, instead of modeling this AK itself, such as done with the domain model in our approach.

Besides our quasi-controlled experiment, Falessi et al. (2006) performed an experiment regarding AK. They evaluated whether making decisions, goal, and alternatives explicit in the form of tables improves individual and team decision making. The results of their experiment indicated this indeed seems to be the case.

Another way to formally describe AK is to make use of an Architecture Description Language (ADL) (Medvidovic and Taylor, 2000). ADLs offer a formal model to express certain concepts and relationships. Often the selection of concepts supported are limited to those of the component & connector (Clements et al., 2002) view. The Knowledge Architect could be extended with a domain model describing the concepts and relationships found in ADLs, thereby making integration with existing ADLs possible.

9. Limitations

The presented method and the supporting tool is based on codifying AK by enriching architecture documentation with formalized AK. Despite the benefits of resolving the challenges identified in Section 2, the approach suffers from the fundamental limitations of AK codification:

- **Cost.** The biggest and foremost limitation is the cost of *capturing and maintaining* the AK by means of a formalism. Most knowledge management approaches assume that the knowledge is already formalized and readily available. However, in practice this is often not the case for AK. Hence, minimizing the cost of capturing and maintaining AK is as important as maximizing the benefits, which formalized AK offers. Our approach attempts to minimize this cost by offering *integrated* tool support that automatically reuses as much context as possible. For example, with the Word plug-in the user does not have to retype a description, simply selecting the text is good enough. However the cost of annotating an architecture document of a large and complex system, remains substantial.
- **Start-up problem.** For most approaches, formalization of knowledge only starts to provide tangible benefits once a significant part of the knowledge has been formalized (Horner and Atwoord, 2006). Consequently, in the initial stages of knowledge capturing, no benefits are perceived from the point of view of knowledge creators. This in turn discourages people from capturing knowledge, which leads to less formalized knowledge and less benefits. Hence, an approach that is incremental in nature is needed. With the Document Knowledge Client we offer such an incremental approach, since a software architecture document does not have to be completely annotated to start having (limited) benefits from these annotations. This partially solves the start-up problem but does not eliminate it.
- **Asymmetric benefit.** The people who capture AK during the architecting process (i.e. the producers) are often not the people using this knowledge (i.e. the consumers). Formalized AK can easily provide benefits for consumers. However, the producers do not usually perceive direct benefits for themselves. This results in an asymmetric benefit between producers and consumers of AK, and therefore lack of motivation for producers to capture complete and high quality AK. Hence, a good codification approach should not only offer benefits for the consumers of the knowledge, but also for the producers. With the Document Knowledge Client we offer such benefits through the different types of checking (correctness, completeness and consistency). These features help architects writing a software architecture by reminding them which parts of the architecture still require further attention. However providing motivation for knowledge producers also largely depends on organizational and process issues.

Table A.1
Ratings of reviewer comments of the first hour.

Subject		1			2		3			4		5			6		7		8		9		10		11		12		13		14		15		16	
	Chapter 1																																		
Situation		3			1		2			1		3			1		1		3		3		1		2		2		2		2		1		3	
Comment weight	1	0	0	0	0	0	2	0	0	0	1	1	3	0	2	0	0	0	0	2	1	0	0	0	3	1	4	0	0	0	0	1	2		
	2	1	2	0	1	4	2	0	0	5	1	0	1	0	3	1	0	0	0	1	1	4	1	2	6	8	3	2	1	0	1	9	7		
	3	1	1	1	1	0	0	1	1	1	4	4	1	6	4	0	0	0	0	1	2	0	2	4	2	3	6	4	4	0	0	9	8		
	4	1	0	1	0	0	0	0	0	0	0	0	0	2	0	1	1	0	0	0	0	0	1	7	2	2	0	1	2	1	0	1	3		
	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0		
	Chapter 2 and 3																																		
Situation		2			3		1			3		2			2		2		2		2		3		1		1		3		1		3		1	
Comment weight	1	0	0	2	4	1	1	0	1	0	1	0	0	0	1	0	0	1	1	0	2	0	0	1	3	2	6	1	3	0	0	3	12		
	2	0	0	6	3	5	2	1	2	1	0	2	1	0	1	0	0	0	0	3	3	1	0	3	10	4	5	5	3	1	1	5	11		
	3	0	0	0	2	3	4	1	0	1	3	0	2	2	0	0	0	0	0	2	0	0	0	6	2	6	0	6	5	1	1	12	2		
	4	1	1	1	0	2	3	0	0	2	0	1	0	0	0	0	0	0	0	0	0	0	1	3	1	1	2	0	1	0	0	5	0		
	5	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0		

10. Future work

In this paper, we proposed an approach for enriching architecture documentation with formal AK. This approach addresses to some extent the challenges that current software architecture documentation approaches face: understandability, locating relevant AK, traceability, change impact analysis, design maturity assessment and trust. We illustrated the approach through a large industrial example, by following the method activities and demonstrating the corresponding tool support. Using a quasi-controlled experiment we presented evidence on how the proposed approach helps to tackle one of the challenges: understandability. Based on our experience from LOFAR, the associated ongoing empirical research, and the development of the knowledge architect tooling we see several directions for future work.

In this paper, the focus was on a subset of activities of our generic method (see Section 3). This limited focus left out two important topics concerning documentation enrichment with AK: integration (activity 5) and evolution (activity 6). For the first activity, we already have started to investigate the possibility of integrating different domain models as a way to integrate AK from different sources (Liang et al., 2008). As for the second activity (evolution of AK), we have already combined the knowledge repository of our tool suite with a version management system that records the evolution of individual AK entities. The initial results for both the integration and the evolution activities look promising, but need to be validated with industrial case studies.

Another direction for future work is to investigate ways to improve the search functionality within the Document Knowledge Client. Currently, relating Knowledge Entities in the tool is based on keyword searches and concept classifications. The results of this search might be improved by using information extraction techniques (Cowie and Lehnert, 1996) that make use of the context of the search, i.e. the Knowledge Entity to be related, its position inside the document, and relationships already defined to other KEs. Furthermore we are looking into ways to make the Knowledge Architect interoperable with other AK management tools, e.g. the just-in time AK sharing portal (Farenhorst et al., 2008), in order to make more and different kinds of AK available.

In the quasi-controlled experiment, we left out the situation in which subjects could both consume existing formal AK and produce their own. It is interesting to investigate whether this situation is an improvement over the ones presented in this paper. Another direction for the experiment is to replicate it, as to create more samples. This will allow for stronger statistical evidence and testing of the assumption we made in this paper about the normal distribution of our samples. Moreover, we plan to use the data from the experiment in order to investigate how people annotate AK inside documents and therefore better understand the process of producing formal AK. An interesting research question in this respect is whether there exist differences in the way people annotate a software architecture document and how that affects the produced AK.

Acknowledgements

This research has been sponsored by the Dutch Joint Academic and Commercial Quality Research & Development (Jacquard) program on Software Engineering Research via contract 638.001.406 GRIFFIN: a GRId For inFormatIoN about architectural knowledge. We would like to thank the people from ASTRON who participated in this research, in particular, Kjeld van der Schaaf. Special mention goes to the master students Jens Rasmussen, Natasja Sterenborg, Hubert ten Hove, Alex Haan, Joris Best, and Marco van der Kooi for their work on the various parts of the Knowledge Architect tool suite. In addition, we would like to thank Industrial Software Systems of ABB Corporate Research for providing the opportunity to revise this publication.

Appendix A

See Table A.1.

References

Antoniou, G., van Harmelen, F., 2004. A Semantic Web Primer. MIT Press.
Avgeriou, P., Kruchten, P., Lago, P., Grisham, P., Perry, D., 2007. Architectural knowledge and rationale: issues, trends, challenges. ACM SIGSOFT Software Engineering Notes 32 (4), 41–46.
Avgeriou, P., Lago, P., Kruchten, P., 2008. Third international workshop on sharing and reusing architectural knowledge (SHARK 2008). ICSE Companion, pp. 1065–1066.
Babar, M., Gorton, I., Kitchenham, B., 2006. A framework for supporting architecture knowledge and rationale management. In: Dutoit, A.H., McCall, R., Mistrík, I., Paech, B. (Eds.), Rationale Management in Software Engineering. Springer-Verlag, pp. 237–254 (Chapter 11).
Babar, M.A., de Boer, R.C., Dingsøyr, T., Farenhorst, R., 2007. Architectural knowledge management strategies:approaches in research and industry. In: Proceedings of the 2nd Workshop on SHAring and Reusing architectural Knowledge – Architecture, Rationale, and Design Intent (SHARK/ADI 2007).
Bass, L., Clements, P., Kazman, R., 2003. Software architecture in practice, second ed. Addison Wesley.
Beck, K., Fowler, M., 2000. Planning Extreme Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
Broekstra, J., Kampman, A., van Harmelen, F., 2002. Sesame: a generic architecture for storing and querying rdf and rdf schema. In: Proceedings of the First International Semantic Web Conference on The Semantic Web (ISWC 2002). Springer-Verlag, London, UK, pp. 54–68.
Butcher, H.R. (2004). Lofar: first of a new generation of radio telescopes. In: Proceedings of SPIE.

Capilla, R., Nava, F., Pérez, S., Duenas, J.C., 2006. A web-based tool for managing architectural design decisions. SIGSOFT Software Engineering Notes 31 (5).
Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J., 2002. Documenting Software Architectures. Views and Beyond. Addison Wesley.
Cowie, J., Lehnert, W., 1996. Information extraction. Communication of ACM 39 (1), 80–91.
de Boer, R.C., van Vliet, H., 2008. Architectural knowledge discovery with latent semantic analysis: constructing a reading guide for software product audits. Journal of Systems and Software 81 (9), 1456–1469.
de Boer, R.C., Farenhorst, R., Lago, P., van Vliet, H., Jansen, A.G.J., 2007. Architectural knowledge: Getting to the core. In: Proceedings of the Third International Conference on the Quality of Software Architectures (QoSA 2007), LNCS, vol. 4880, pp. 197–214.
de Holan, P.M., Phillips, N., 2004. Organizational forgetting as a strategy. Strategic Organization 2 (4), 423–433.
Duenas, J.C., Capilla, R., 2005. The decision view of software architecture. In: Morrison, R., Oquendo, F. (Eds.), EWSA, Lecture Notes in Computer Science, vol. 3527. Springer, pp. 222–230.
Falessi, D., Cantone, G., Becker, M., 2006. Documenting design decision rationale to improve individual and team design decision making: an experimental evaluation. In: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE '06). ACM Press, New York, NY, USA, pp. 134–143.
Farenhorst, R., Izaks, R., Lago, P., van Vliet, H., 2008. A just-in-time architectural knowledge sharing portal. Wicsa, 125–134.
Handschuh, S., Staab, S., Ciravegna, F., 2002. S-cream – semi-automatic creation of metadata. In: EKAW '02: Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web. Springer-Verlag, London, UK, pp. 358–372.
Hansen, M.T., Nohria, N., Tierney, T., 1999. What's your strategy for managing knowledge? Havard Business Review 77 (2), 106–116.
Hofmeister, C., Nord, R., Soni, D., 2000. Applied Software Architecture. Addison Wesley.
Hofmeister, C., Nord, R.L., Soni, D., 2005. Global analysis: moving from software requirements specification to structural views of the software architecture. IEE Proceedings Software (4), 187–197.
Horner, J., Atwoord, M., 2006. Effective design rationale: understanding the barriers. In: Dutoit, A.H., McCall, R., Mistrik, I., Paech, B. (Eds.), Rationale Management in Software Engineering. Springer-Verlag, pp. 73–88 (Chapter 3).
IEEE/ANSI, 2000. Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Standard No. 1471-2000, Product No. SH94869-TBR.
Jansen, A.G.J., Bosch, J., 2005. Software architecture as a set of architectural design decisions. In: Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA 2005). pp. 109–119 (November).
Jansen, A., de Vries, T., Avgeriou, P., van Veelen, M., 2008a. Sharing the architectural knowledge of quantitative analysis. In: Proceedings of the Fourth International Conference on the Quality of Software-Architectures (QoSA 2008). LNCS, vol. 5281, pp. 220–234.
Jansen, A.G.J., Bosch, J., Avergiou, P., 2008b. Documenting after the fact: recovering architectural design decisions. Journal of Systems and Software 81 (4), 536–557 (April).
Jedlitschka, A., Pfahl, D., 2005. Reporting guidelines for controlled experiments in software engineering. 2005 International Symposium on Empirical Software Engineering, 10.
Kahan, J., Koivunen, M.-R., 2001. Annotea: an open rdf infrastructure for shared web annotations. In: WWW '01: Proceedings of the 10th International Conference on World Wide Web. ACM, New York, NY, USA, pp. 623–632.
Kiryakov, A., Ognyanov, D., Manov, D., 2005. Owlim – a pragmatic semantic repository for owl. In: Proceedings of the Web Information Systems Engineering Workshop (WISE). LNCS, vol. 3807, pp. 182–192.
Kruchten, P., 1995. The 4+1 view model of architecture. IEEE Software 12 (6), 42–50.
Kruchten, P., 2000. The Rational Unified Process: An Introduction, second ed. Addison-Wesley.
Kruchten, P. 2004. An ontology of architectural design decisions in software intensive systems. In: 2nd Groningen Workshop on Software Variability. pp. 54–61.
Kruchten, P., Lago, P., van Vliet, H. 2006. Building up and reasoning about architectural knowledge. In: Proceedings of the Second International Conference on the Quality of Software Architectures (QoSA 2006).
Lago, P., Avgeriou, P., 2006. First workshop on sharing and reusing architectural knowledge. SIGSOFT Software Engineering Notes 31 (5), 32–36.
Lago, P., Avgeriou, P., Capilla, R., Kruchten, P., 2008. Wishes and boundaries for a software architecture knowledge community. In: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008). IEEE Computer Society, Washington, DC, USA, pp. 271–274.
Lethbridge, T., Singer, J., Forward, A., 2003. How software engineers use documentation: the state of the practice. Software, IEEE 20 (6), 35–39.
Letsche, T.A., Berry, M.W., 1997. Large-scale information retrieval with latent semantic indexing. Information Science 100 (1–4), 105–137.
Liang, P., Jansen, A., Avgeriou, P., 2008. Sharing architecture knowledge through models: quality and cost. The Knowledge Engineering Review 23 (2).
Lofar website, Lofar project website. http://www.lofar.org/.
Medvidovic, N., Taylor, R.N., 2000. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26 (1), 70–93.
Nonaka, I., Takeuchi, H., 1995. The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation. Oxford University Press Inc., USA.
O'Hara, K., Sellen, A., 1997. A comparison of reading paper and on-line documents. In: CHI '97: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM, New York, NY, USA, pp. 335–342.
Rodgers, J.L., Nicewander, W.A., 1988. Thirteen ways to look at the correlation coefficient. The American Statistician 42 (1), 59–66.
Schwaber, K., Beedle, M., 2001. Agile Software Development with SCRUM. Prentice Hall.
Sheskin, D.J., 2003. Handbook of Parametric and Nonparametric Statistical Procedures, third ed. Chapman & Hall/CRC.
Tang, A., Babar, M.A., Gorton, I., Han, J., 2005a. A survey of the use and documentation of architecture design rationale. In: Proceeding of the Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA 2005). pp. 89–99.
Tang, A., Jin, Y., Han, J., Nicholson, A.E., 2005b. Predicting change impact in architecture design with bayesian belief networks. In: Proceeding of the Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA 2005). IEEE Computer Society, pp. 67–76.
Tang, A., Jin, Y., Han, J., 2007. A rationale-based architecture model for design traceability and reasoning. Journal of Systems and Software 80 (6), 918–934.
Tyree, J., Akerman, A., 2005. Architecture decisions: Demystifying architecture. IEEE Software 22 (2), 19–27.
van der Ven, J.S., Jansen, A.G.J., Avgeriou, P., Hammer, D.K., 2006a. Using architectural decisions. In: Second International Conference on the Quality of Software Architecture (Qosa 2006).
van der Ven, J.S., Jansen, A.G.J., Nijhuis, J.A.G., Bosch, J., 2006b. Design decisions: The bridge between rationale and architecture. In: Dutoit, A.H., McCall, R., Mistrik, I., Paech, B. (Eds.), Rationale Management in Software Engineering. Springer-Verlag, pp. 329–348 (Chapter 16).
Vargas-Vera, M., Motta, E., Domingue, J., Lanzoni, M., Stutt, A., Ciravegna, F., 2002. Mnm: ontology driven semi-automatic and automatic support for semantic markup 2473, 213–221.
W3C, 2004. Owl web ontology language – w3c recommendation. <http://www.w3.org/TR/owl-features/>.
Zimmermann, O., Zdun, U., Gschwind, T., leymann, F., 2008. Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method. In: Proceedings of the seventh Working IEEE/IFIP Conference on Software Architecture (WICSA). IEEE Computer Society, Los Alamitos, CA, USA, pp. 157–166.

Anton Jansen is a scientist at ABB corporate research in the Software Architecture & Usuability (SARU) group in Västerås Sweden since 2009.

Between 2002 and 2009, he was a member of the Software Engineering and Architecture (SEARCH) research group at the University of Groningen, the Netherlands. He received a master of science degree in computing science in 2002, as well as a Ph.D. in Software Architecture in 2008 from the University of Groningen, the Netherlands. He has worked as a Ph.D. associate (2002–2006) on architectural decisions under supervision of Jan Bosch, and as a postdoc (2006–2008) on the Dutch Joint Academic and Commercial Quality Research & Development (Jacquard) project GRIFFIN.

His research interests are software architecture and architectural knowledge.

Paris Avgeriou holds the chair of Software Engineering, at the Department of Mathematics and Computing Science, University of Groningen, the Netherlands. He is the head of the Software Engineering and Architecture (SEARCH) research group since September 2006. He received a diploma (M.Sc.) in Electrical and Computer Engineering (1999), as well as a Ph.D. in Software Engineering (2003) from the National Technical University of Athens (NTUA), Greece. He has worked as a Senior Researcher at the Fraunhofer IPSI, Darmstadt, Germany (2005), and the Laboratory for Advanced Software Systems, University of Luxembourg (2004), under the Fellowship Programme of ERCIM, as a visiting Lecturer at the Department of Computer Science (2003), University of Cyprus, and as a research and teaching assistant at NTUA (1999–2002). He has been leading a number of research projects on software architecture, has an extensive publication list and is involved in the organization of conferences and workshops. He has received awards and distinctions for both teaching and research. He is a member of IEEE, ERCIM, Hillside Europe and a founding member of the World-Wide Institute of Software Architects. His research interests concern the area of software architecture and particularly architecture knowledge, patterns, and processes.

Jan Salvador van der Ven is an ICT Consultant at Ordina Oracle Solutions since 2006. He conducted research at the University of Groningen, the Netherlands, from 2003 to 2006, under supervision of Jan Bosch. Before that, he received a master of science degree in Computing Science with specialization in Software Engineering in 2003 from the same university.

During his Ph.D. research, he participated in the SeCSE (Service Centric Systems Engineering) and GRIFFIN (a GRId For inFormatIoN about architectural knowledge) projects. His research focused on the explicit modeling of architecture decisions during the construction and evolution of systems. His current research interest concerns the need for light-weight software architectures in agile software development.